THIS MONTH’S COLUMN features Vaughn Vernon, a consultant and mentor with more than 25 years’ experience in software design, development, and architecture. He’s also the author of Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka (Addison-Wesley, 2016). The reactive model is an emerging topic in software engineering, driven by user sensitivity to latency and the desire to use computing resources more efficiently. In his book, Vernon explains how these two forces have converged on a model organized around events rather than blocking threads.

Topics not covered here because of space include a typical approach for designing a system made of actors and the role of enterprise integration patterns.

Please contact us with your ideas and thoughts about this column, or any aspect of the show, at team@se-radio.net.
—Robert Blumen

Stefan Tilkov (ST): What does reactive programming mean? Is it something new, or have we been doing it for a long time?

Vaughn Vernon (VV): It’s both new and old. Reactive programming on the server side is going through a repopularization; on the client side, it’s been going on for a long time. As far back as 1973, [computer scientist] Carl Hewitt was working on an experimental model called the actor model. Although the computing power available at the time wasn’t suited for parallel or concurrent processing, he still formulated this model.

Moving forward to the late 1980s and early 1990s, GUI programming was being developed. The Windows API and X-Windows were using reactive programming, although it probably wasn’t called that at the time. The idea is that a user would move the mouse or click a command button that created events that were sent to the program, and the program would react to those events.

A lot of people are doing reactive programming now, especially in modern Web development. If you’ve ever written JavaScript that reacts to a mouse movement or button click, that’s reactive programming.

Reactive programming on the server side is a bit different from the client side. It uses the same basic principles, but components on the server side are reacting to some sort of stimulus rather than just the UI.

ST: How does the actor model work?

VV: The actor model focuses on an actor as the central and primary unit of computation. You might think of an actor as...
a single-responsibility computational unit that’s responsible for one thing specifically. It knows how to handle, or react to, some set of incoming messages. As a reaction to that stimulus, it might change its internal state, it might send a message back to the sender of the message that it’s currently handling, or it might send out messages to other actors as outgoing messages.

The actor does not have to be concerned about the typical concurrency problems that are faced by various other styles of computing where we’re using multithreading across any number of components, because the actor is receiving the messages asynchronously and one at a time. It never has to worry about two threads trying to modify their state at the same time; there will only be one thread assigned to the actor at any given time.

This means the actor must share nothing about its internal state. It must keep its internal state to itself. Because the object is handling one message at a time, the handler can operate across the entire state of the actor without having to worry about race conditions.

You could also think of the actor as being a perfect transactional boundary. Everything that happens within the actor, as it’s dealing with one message at a time, is a perfect boundary for transactional control. And no other actors will be influenced directly by what happens to the actor’s state at that time, so it’s a great transactional boundary as well.

ST: It sounds like an active object. It has some of the properties of objects but also has a thread of control?

VV: That’s a good way to put it. Think of an actor as an object with public methods that can be invoked by other objects. An actor is similar to an object, except when you send an actor a message, it doesn’t immediately invoke a method on the actor. The message is received asynchronously from the message send itself, so the message will be handled on a different thread inside the actor. There are asynchronous computing and concurrency going on around the actor, but inside the actor it can call its own methods synchronously. It’s sort of like objects with asynchronous messaging.

ST: What’s the actor model’s relation to Scala and Akka?

VV: After the actor model was introduced in 1973, its use was quite limited, primarily because there weren’t a lot of cores on individual machines. At that time it was more about scaling up than scaling out.

Taking a leap forward to when the actor model was more widely in use, [you’d see the influence of] the Erlang programming language. Erlang has become a more popular language of late, but I would say it’s still not a completely mainstream language like Java or C#. But if you look at Scala and Akka from the point of view of someone who knows Erlang, you can see a lot of similarities. An Erlang-flavored language with actors on the JVM [Java virtual machine]—that’s what Scala and Akka give you.

ST: How would you compare the effects of those approaches? What’s the downside, and what are each model’s benefits?

VV: In object-oriented programming, you have an object. You can invoke a method on it. [Calling it] a method invocation on an object is the more modern way to talk about it because that’s how it actually works in Java or C#. In Smalltalk
(believe it or not, Smalltalk is a language based on message sending), I literally send a message to an object. The method that gets invoked on the object is a mapping of the message send to a method selector internal to the object. By default, in Smalltalk each message to an object is handled on the same thread as the invocation or the message; still, there is this idea of a separation between the external object that is sent (the message) and the internal object that receives it. Thinking about it that way helps us take one step forward toward the actor model and even concurrent programming.

If you think about that same Smalltalk application where I send the message to an object, then think of the messaging as happening asynchronously. There is [an inbound message queue] that receives the invocation request, but the [method] invocation won’t occur until that object has an opportunity to accept the latest message and react to it. That is a stepwise refinement toward the actor model.

On the other hand, if you think about the kinds of multithreaded programming APIs that are available to us (such as the concept of a thread in Java), it’s the programmer’s responsibility to create a thread, to start the thread running, and to somehow receive input to that thread. And how do we receive input? [That’s left up] to the application designer.

Because there’s no way to automatically prevent that thread from receiving input from multiple threads at one time, we must worry about synchronization. [That raises issues such as,] Do we have full rights to the data that thread has within its object and its object state at any given time? Is it possible for that thread to receive multiple input requests simultaneously? Do we synchronize this method, or do we synchronize this hot block of code within a method? How do we synchronize it? How do we make sure that we don’t turn our multithreaded program back into a glorified single-threaded program because we are just so poor at writing multithreaded code? What we end up doing is blocking everything around some critical sections of code.

With the actor model, we don’t have those concerns because the actor is the central logical unit of computation. There is a scheduler that understands that when the actor receives a message, it must, at some point, receive a thread or be given a thread to run on for some period of time. The scheduler might allow the actor to run that single message it received. If there are five messages in queue for the actor, it might allow it to run all five of the messages before it takes the thread away from the actor and gives it to another actor. If you look at the internals of how an actor model is created or designed, you would probably see a thread pool where the pool has a number of threads that can optimally be used by the application at any given time.

ST: What you’re saying, I think, is that the actor runtime takes care of mapping the application-level concept of an actor to the OS or runtime-level concept of a thread. So, as an application programmer, I don’t have to do that myself; I can focus just on building actors that will be mapped to the underlying resources as efficiently as the runtime can make it.

VV: Exactly. Programming with actors is different. As soon as you start to deal with concurrency and you realize, “I’ve sent a message to an actor, but the response I get back will not cause me to block until I receive that response,” you can do something else during that time or give up the processor and allow someone else to use it—basically, do nothing. I just finish the handling of the message that I’ve currently received and then I return it. When I return it, a thread on a core gets handed to another actor to use. As soon as you realize you’re not doing synchronous programming anymore, it changes your mindset. It’s good to learn to think that way because it allows the actor model to perform as well as it possibly can on any given machine at any given time, with the numbers of actors that have to process messages.

ST: Am I exchanging one way of doing things for another or exchanging one level of complexity for another? With the actor model, do I have to learn a new set of patterns that are just as good or bad as the patterns I’m used to when using the thread model? Or is it a different category of patterns?

VV: It’s a different category of patterns. If you’ve ever used any sort of messaging middleware, you know that there’s a difference between synchronous and asynchronous programming. While it’s different and introduces a new set of challenges, it also has to introduce a new set of benefits; otherwise, we wouldn’t use it. Consider a Web application handling a request. We delegate from a controller to the application layer and from the application layer to the domain layer, and the domain layer will cause some sort of persistence interactions and so forth.

What we have to recognize is that the thread that was assigned at the
beginning is 100 percent used by that one single request that came from the Web. Let’s pick on the database for a moment. While we’re waiting for the database to do something, that thread is blocking and it’s waiting; it might wait for seconds at a time. We only have a finite number of threads. If we have 16 cores with one thread per core, then that entire powerful machine that’s running might be blocking 16 different times on the database. Nothing else is happening on that machine for that period of time because it can’t happen.

On the other hand, in the actor model, instead of blocking, we could have sent a message to a database that we want something. We know that, in some period of time, the database will send a message back to us. We’re not blocking; we’re simply inactive—and the thread that would have been assigned to us in a normal n-tier Web application is now in use by some other thread that can move forward. We could handle dozens or hundreds more application-level requests from the Web tier than we could in that same period of time with just the 16 threads that are going to end up being blocked at the database, instead of being used by other actors at the same time.

That’s where the real payoff comes from. The problem we’re now facing with computing is that processor and clock speeds have fallen off for about 10 years. Processors are not getting any faster, at least not exponentially faster as they were in previous decades. As processor performance is flattening out, what do we do? If we’re still handling only 16 requests at one time, the processor is just sitting there, doing nothing, while these 16 different Web requests are blocking for some sort of database response. With the actor model, we’re simultaneously moving other things forward with the same processor.

If you’ve ever written JavaScript that reacts to a mouse movement or button click, that’s reactive programming.

STEFAN TILKOV is cofounder and principal consultant at innoQ. Contact him at stefan.tilkov@innoq.com.

See www.computer.org/software-multimedia for multimedia content related to this article.


Postmaster: Send undelivered copies and address changes to IEEE Software, Membership Processing Dept., IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854-4141. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8, Canada. Printed in the USA.

Reuse Rights and Reprint Permissions: Educational or personal use of this material is permitted without fee, provided such use: 1) is not made for profit; 2) includes this notice and a full citation to the original work on the first page of the copy; and 3) does not imply IEEE endorsement of any third-party products or services. Authors and their companies are permitted to post the accepted version of IEEE-copyrighted material on their own web servers without permission, provided that the IEEE copyright notice and a full citation to the original work appear on the first screen of the posted copy. An accepted manuscript is a version which has been revised by the author to incorporate review suggestions, but not the published version with copyediting, proofreading, and formatting added by IEEE.

For more information, please go to: http://www.ieee.org/publications_standards/publications/rights/paperversionpolicy.html. Permission to reprint/republish this material for commercial, advertising, or promotional purposes or for creating new collective works for resale or redistribution must be obtained from IEEE by writing to the IEEE Intellectual Property Rights Office, 445 Hoes Lane, Piscataway, NJ 08854-4141 or pubs-permissions@ieee.org. Copyright © 2016 IEEE. All rights reserved.

Abstracting and Library Use: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy for private use of patrons, provided the per-copy fee indicated in the code at the bottom of the first page is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923.